Exercise 1 – Prefix, subword and suffix of a word

Recall that a word is a finite sequence of characters and that one writes \( w_1 \cdot w_2 \) for the word defined as the concatenation of two words \( w_1 \) and \( w_2 \). We say that a word \( u \) is a prefix of a word \( v \) when \( v = u \cdot w \) for some words \( w \), that \( u \) is a subword of a word \( v \) when \( v = w_1 \cdot u \cdot w_2 \) for some words \( w_1 \) and \( w_2 \), and that \( u \) is a suffix of a word \( v \) when \( v = u \cdot w \) for some words \( w \).

§1.a. Define a Haskell function

\[
\text{isPrefix} :: \text{String} \to \text{String} \to \text{Bool}
\]

which tests whether the first input of the function is prefix of the second input:

\[
\begin{align*}
\text{ghci} &> \text{isPrefix } "\text{gene}" \ "\text{_generous}" \\
\text{} &> \text{True} \\
\text{ghci} &> \text{isPrefix } "\text{genetic}" \ "\text{generous}" \\
\text{} &> \text{False}
\end{align*}
\]

§1.b. Define a function

\[
\text{isSubword} :: \text{String} \to \text{String} \to \text{Bool}
\]

which tests whether the first input of the function is subword of the second input:

\[
\begin{align*}
\text{ghci} &> \text{isSubword } "\text{nero}" \ "\text{generous}" \\
\text{} &> \text{True} \\
\text{ghci} &> \text{isSubword } "\text{neros}" \ "\text{generous}" \\
\text{} &> \text{False}
\end{align*}
\]

§1.c. Using \text{isPrefix} function defined in §1.a. together with the \text{reverse} function of the Haskell standard Prelude library

\[
\text{reverse} :: \text{String} \to \text{String}
\]

which reverses a string, define a function

\[
\text{isSuffix} :: \text{String} \to \text{String} \to \text{Bool}
\]
which tests whether the first input of the function is suffix of the second input:

```
ghci> isSuffix "erous" "generous"
True
ghci> isSuffix "prosperous" "generous"
False
```

§1.d. Define a function

```
isSuffix2 :: String -> String -> Bool
```

with just the same behavior as the function `isSuffix` defined in §1.c. except that the `isSuffix2` function is implemented this time without the `reverse` function.

**Exercise 2 – Binary Search Trees**

In this exercise, a *key* is defined as an integer, and *binary tree* is either empty or a node labelled with a key together with a pair of binary trees, called the left and right subtrees of the node. The type `Tree` of binary trees is thus defined in the following way in Haskell:

```
data Tree = Nil | Node Int Tree Tree
```

A typical binary tree may be thus depicted as follows:

```
  5
 / 
7   2
/ 
1   3
```

and represented by the Haskell expression of type `Tree`:

```
tree = Node 5 (Node 7 Nil Nil) (Node 2 (Node 1 (Node 5 Nil Nil) Nil) (Node 3 Nil Nil))
```

Note that every node `n` in a binary tree induces a left subtree noted `n >>= left` and a right subtree noted `n >>= right`. A *binary search tree* (BST) is defined as a binary tree satisfying the additional property that for every node `n` of the binary tree:
• the key of \( n \) is larger (or equal) than every key in the left subtree \( n \triangleright \text{left} \),

• the key of \( n \) is smaller (or equal) than every key in the right subtree \( n \triangleright \text{right} \).

A typical illustration of a binary search tree is the following binary tree:

\[
\text{bstree} = \begin{array}{c}
2 \\
/ \ \\
1 \quad 5 \\
/ \quad / \\
3 \quad 5 \quad 7 \\
/ \\
3 \\
\end{array}
\]

and represented by the Haskell expression of type \text{Tree}:

\[
\text{bstree} = \text{Node} 2 (\text{Node} 1 \text{Nil} \text{Nil}) (\text{Node} 6 (\text{Node} 5 (\text{Node} 3 \text{Nil} \text{Nil}) \text{Nil}) (\text{Node} 7 \text{Nil} \text{Nil}))
\]

§2.a. Define a function

\[
\text{isBST} :: \text{Tree} \rightarrow \text{Bool}
\]

which tests whether a binary tree is a binary search tree:

\[
\text{ghci}> \text{isBST} \text{tree} \\
\text{False} \\
\text{ghci}> \text{isBST} \text{bstree} \\
\text{True}
\]

§2.b. Define a function

\[
\text{contains} :: \text{Int} \rightarrow \text{Tree} \rightarrow \text{Bool}
\]

which tests whether a specific number given as input appears as a key of one node in the binary search tree:

\[
\text{ghci}> \text{contains} 3 \text{bstree} \\
\text{True} \\
\text{ghci}> \text{contains} 6 \text{bstree} \\
\text{False}
\]

§2.c. Define a function
**insert :: Int -> Tree -> Tree**

which inserts a new node in a binary search tree, in such a way that the resulting binary tree is still a binary search tree. Typically, inserting a node with key 6 in the binary search tree `bstree` above will produce the binary search tree below:

![Binary search tree with 6 inserted]

§2.d. Define a function

**extract :: Tree -> (Int, Tree)**

which finds the smallest key in a non-empty binary search tree, and removes the corresponding node in such a way as to produce a binary search tree. Typically, extracting the smallest key from the binary search tree `bstree` above produces the integer 1 together with the binary search tree obtained by removing the « leftmost » node:

![Binary search tree with smallest key extracted]

Repeating the extraction algorithm will produce the integer 2 together with the binary search tree obtained by removing the root node, which may be seen as the the « leftmost » node of the original tree:

![Binary search tree with root node removed]
§2.e. Using the `extract` function just defined in §2.d, define a function

\[
\text{removeroot :: Tree -> Tree}
\]

which removes the root of a non-empty binary search tree and returns the resulting binary search tree. Typically, applying the function to the binary search tree `bstree` above will produce the binary search tree below:

```
  3
 / \
1   5
 / \   \
5   7
```

applying the removal algorithm another time will produce the binary search tree below:

```
  5
 /   \
1     7
```

then applying the algorithm once again will produce the binary search tree below:

```
  5
 /   \
1     7
```

§2.f. Using the `removeroot` function just defined in §2.e, define the function

\[
\text{delete :: Int -> Tree -> Tree}
\]

which given an integer \( k \) and a binary search tree `bst` returns:

1. the binary search tree `bst` itself — when the key \( k \) does not appear in the tree `bst`,
2. a binary search tree obtained by removing a specific node with key \( k \) from the binary search tree `bst` — when the key \( k \) appears in the tree `bst`. 
Typically, applying the \texttt{delete} function to 5 and to the binary search tree \texttt{bstree} above will produce the binary search tree below:

```
        2
       /|
      1 7
     /|
    5 3
```

Then, applying the \texttt{delete} function to 2 and to the resulting binary search tree will produce the binary search tree below:

```
        3
       /|
      1 7
     /|
    5
```

**Exercise 3 – A type for mathematical expressions**

We introduce the following type of mathematical expressions

```haskell
data Expr = Num Integer
          | Add Expr Expr
          | Mult Expr Expr deriving Show
```

For example, the mathematical expression $1 + (2 \times 3)$ can be written as the expression

```
Add (Num 1) (Mult (Num 2) (Num 3))
```

§3.1. Define an \texttt{eval} function of type

```
eval :: Expr -> Integer
```

which evaluates the arithmetic value of an expression:

```
ghci> eval (Add (Num 1) (Mult (Num 2) (Num 3)))
7
```
§3.2. Define a `prettyprint` function

```haskell
prettyprint :: Expr -> String
```

Typically, the `prettyprint` function should behave in the following way:

```haskell
ghci> prettyprint (Add (Num 1) (Mult (Num 2) (Num 3)))
"(1+(2 ∗ 3))"
```

§3.3. Explain how to see the type `Expr` of mathematical expressions as an instance of the type class `Show`, in such a way that the `show` function is not derived anymore from the definition of the type `Expr`, as we did at the beginning of the exercise, but defined as the `prettyprint` function implemented in §3.2. Once this instantiation of the class type `Show` performed, the interactive mode of `ghc` should behave in the following way:

```haskell
ghci> let expression = Add (Num 1) (Mult (Num 2) (Num 3))
ghci> expression
"(1+(2 ∗ 3))"
```

**Exercise 4 [optional] – A parameterized type for mathematical expressions**

We want to generalize the situation of §3. by introducing the following parameterized type of mathematical expressions

```haskell
data Expr a = Num a
            | Add (Expr a) (Expr a)
            | Mult (Expr a) (Expr a) deriving Show
```

where `Expr a` is a type parameterized by the type `a`.

§4.1. Describe in that context the type of the expression

```
Add (Num 1) (Mult (Num 2) (Num 3))
```

which you are welcome to check in the interactive mode of `ghc`.

§4.2. Explain why the `prettyprint` function defined in §3.2 and adapted to this situation has parameterized type

```haskell
prettyprint :: Show a => Expr a -> String
```

as you are welcome to check in the interactive mode of `ghc`.

§4.3. Do you see any good reason (and tentative application) for adopting the parameterized definition of `Expr` instead of the original one defined at the beginning of §3 and restricted to integer values.